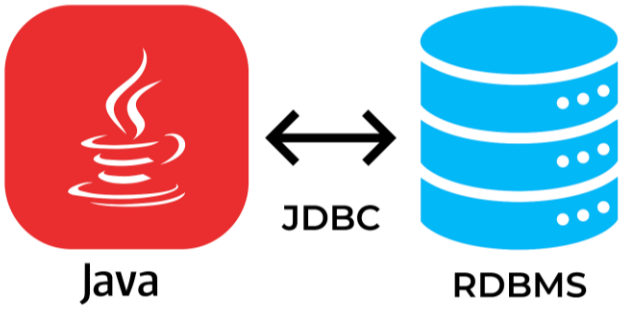
Angel Yeremi Ledesma Ayala

JDBC

¿Por qué JDBC?

JDBC (Java Database Connectivity) desempeña un papel fundamental en el desarrollo de aplicaciones Java que requieren acceso a bases de datos relacionales. Aquí se explican las razones por las cuales los desarrolladores eligen JDBC:

1. Universalidad: JDBC es una API estándar de Java que permite a los desarrolladores escribir aplicaciones Java que se pueden conectar a una variedad de bases de datos relacionales, como MySQL, Oracle, PostgreSQL y más. Esto significa que el mismo código Java puede funcionar con diferentes bases de datos sin necesidad de modificarlo.
2. Independencia de la plataforma: Java es conocido por su capacidad de portabilidad entre diferentes sistemas operativos. JDBC sigue esta filosofía, lo que permite que las aplicaciones desarrolladas en Java se ejecuten en diversas plataformas sin cambios significativos.
3. Seguridad y control: JDBC proporciona un alto nivel de seguridad y control en las operaciones de base de datos. Los desarrolladores pueden establecer conexiones seguras y administrar las operaciones de manera controlada.
4. Escalabilidad: Las aplicaciones Java que utilizan JDBC pueden manejar grandes cantidades de datos y, por lo tanto, son escalables para aplicaciones empresariales y sistemas que requieren el procesamiento de datos a gran escala.
5. Interoperabilidad: JDBC permite la integración de aplicaciones Java con otros sistemas que utilizan bases de datos relacionales, facilitando la interoperabilidad con otras tecnologías y sistemas.
6. Amplia adopción: JDBC ha sido ampliamente adoptado y utilizado en la industria durante muchos años. Esto significa que hay una gran cantidad de recursos, documentación y comunidades de soporte disponibles para los desarrolladores que trabajan con esta tecnología.



Angel Yeremi Ledesma Ayala

Driver Manager

El Driver Manager, o "Administrador de controladores," es una parte clave de JDBC (Java Database Connectivity). Su función principal es gestionar los controladores de JDBC, que son específicos de cada base de datos, y proporcionar conexiones a la base de datos. A continuación, se explica el papel del Driver Manager y su uso en el contexto de JDBC:

1. Carga de controladores (Drivers): Antes de poder interactuar con una base de datos específica, es necesario cargar el controlador JDBC correspondiente. Cada base de datos tiene su propio controlador JDBC, y estos controladores deben estar disponibles en el classpath del proyecto. El Driver Manager se encarga de cargar el controlador JDBC adecuado utilizando el método Class.forName(). Aquí hay un ejemplo de cómo cargar el controlador de MySQL:

```
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

2. Establecimiento de una conexión: Una vez que el controlador está cargado, el Driver Manager puede ayudar a establecer una conexión a la base de datos. Se necesita proporcionar la URL de conexión, el nombre de usuario y la contraseña para acceder a la base de datos. Aquí hay un ejemplo de cómo se realiza esto:

```
String url = "jdbc:mysql://localhost:3306/mi_basededatos";
String usuario = "usuario";
String contraseña = "contraseña";

try {
    Connection conexion = DriverManager.getConnection(url, usuario,
    contraseña);
} catch (SQLException e) {
    e.printStackTrace();
}
```

3. Gestión de conexiones: Es importante recordar que las conexiones a la base de datos deben gestionarse cuidadosamente. Después de realizar las operaciones necesarias en la base de datos, es fundamental cerrar la conexión para liberar recursos. Aquí se muestra cómo se puede realizar:

```
try {
    // Realizar operaciones con la conexión
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        if (conexion != null) {
            conexion.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

El Driver Manager es una parte fundamental en la interacción entre aplicaciones Java y bases de datos a través de JDBC. Facilita la carga de controladores, la conexión a la base de datos y la gestión de conexiones. Recordar manejar las excepciones de manera adecuada es esencial para lidiar con posibles errores durante este proceso.

Connections (Conexiones):

En el contexto de JDBC (Java Database Connectivity), una "Connection" o "Conexión" es un objeto que representa una conexión activa a una base de datos. Las conexiones son fundamentales para interactuar con la base de datos y realizar operaciones como consultas, inserciones, actualizaciones y eliminaciones. Aquí se explica el papel de las conexiones y cómo se utilizan en Java:

1. Establecimiento de una conexión: Para crear una conexión a la base de datos, se utiliza el método DriverManager.getConnection(), como se mencionó en el tema anterior. Aquí se muestra un ejemplo nuevamente:

```
String url = "jdbc:mysql://localhost:3306/mi_basededatos";
String usuario = "usuario";
String contraseña = "contraseña";

try {
    Connection conexion = DriverManager.getConnection(url, usuario, contraseña);
} catch (SQLException e) {
    e.printStackTrace();
}
```

2. Realización de operaciones en la base de datos: Una vez que se tiene una conexión válida, se pueden ejecutar diversas operaciones en la base de datos, como consultas SQL, inserciones, actualizaciones y eliminaciones. Para ello, se utilizan objetos llamados "Statements" (sentencias), que se explicarán en el próximo tema.
3. Gestión de la conexión: Es importante recordar que las conexiones deben gestionarse adecuadamente. Después de completar las operaciones en la base de datos, es fundamental cerrar la conexión para liberar recursos. Esto se hace llamando al método close() en el objeto Connection.

```
try {
    // Realizar operaciones en la base de datos

    // Cerrar la conexión cuando ya no sea necesaria
    conexion.close();
} catch (SQLException e) {
    e.printStackTrace();
}
```

4. Control de transacciones: Las conexiones también permiten controlar transacciones en la base de datos. Puedes iniciar una transacción, confirmarla o revertirla según sea necesario para asegurar la integridad de los datos.

5. Manejo de excepciones: Al interactuar con bases de datos, es crucial manejar las excepciones adecuadamente para lidiar con posibles errores, como problemas de conexión o consultas mal formadas.

En resumen, las conexiones son objetos esenciales en JDBC que permiten a las aplicaciones Java establecer comunicación con bases de datos. Facilitan la realización de operaciones en la base de datos y son cruciales para la gestión de recursos y el control de transacciones. Una vez que se ha completado el trabajo con una conexión, es importante cerrarla de manera adecuada para liberar recursos y evitar posibles problemas.

Statements (Sentencias):

En el contexto de JDBC (Java Database Connectivity), las "Statements" (Sentencias) son objetos que se utilizan para ejecutar instrucciones SQL en una base de datos. Las sentencias permiten a las aplicaciones Java enviar consultas, inserciones, actualizaciones y eliminaciones a la base de datos. A continuación, se explica el papel de las sentencias y cómo se utilizan en Java:

1. Tipos de sentencias: JDBC proporciona varios tipos de sentencias que se adaptan a diferentes tipos de operaciones:
    - Statement: Se utiliza para ejecutar consultas SQL estáticas sin parámetros.
    - PreparedStatement: Se utiliza para ejecutar consultas parametrizadas, lo que permite reutilizar la misma sentencia con diferentes valores.
    - CallableStatement: Se utiliza para invocar procedimientos almacenados en la base de datos.
2. Creación de una sentencia: Para crear una sentencia, se utiliza el objeto Connection para obtener una instancia de la sentencia deseada. Aquí se muestra un ejemplo de cómo crear un PreparedStatement:

```
String consultaSQL = "SELECT * FROM usuarios WHERE nombre = ?";

PreparedStatement sentencia = conexion.prepareStatement(consultaSQL);
```

3. Ejecución de la sentencia: Una vez que se ha creado una sentencia, se pueden establecer los parámetros (si es una PreparedStatement) y ejecutarla utilizando los métodos apropiados, como executeQuery() para consultas de selección o executeUpdate() para sentencias de modificación (inserciones, actualizaciones y eliminaciones).

```
// Establecer un valor para el parámetro
sentencia.setString(1, "nombreUsuario");


// Ejecutar la consulta
ResultSet resultado = sentencia.executeQuery();
```

4. Recuperación de resultados: Si la sentencia es una consulta, el resultado se almacena en un objeto ResultSet, que contiene los datos devueltos por la base de datos. Puedes iterar a través del ResultSet para recuperar los datos.
5. Cierre de la sentencia: Después de usar una sentencia, es importante cerrarla para liberar recursos. Esto se hace llamando al método close() en la sentencia.

```
sentencia.close();
```

Las sentencias son fundamentales para realizar operaciones en una base de datos a través de JDBC. Proporcionan una interfaz para enviar consultas SQL y recuperar resultados. Las PreparedStatement son especialmente útiles cuando se trabajan con consultas parametrizadas, ya que evitan problemas de seguridad y optimizan el rendimiento. La correcta gestión de las sentencias y la liberación de recursos es esencial para el funcionamiento eficiente de una aplicación JDBC.

Insertando filas

En el contexto de JDBC (Java Database Connectivity), la inserción de filas en una base de datos es una operación común y esencial para agregar nuevos registros a una tabla. A continuación, se describe cómo se pueden insertar filas en una base de datos utilizando JDBC en Java:

1. Preparar la sentencia de inserción: Para insertar una fila, se utiliza una PreparedStatement. Debes crear una consulta SQL con marcadores de posición para los valores que se van a insertar. Aquí un ejemplo:

```
String consultaSQL = "INSERT INTO usuarios (nombre, edad)
VALUES (?, ?)";

PreparedStatement sentencia =
conexion.prepareStatement(consultaSQL);
```

2. Establecer valores de los parámetros: A continuación, debes establecer los valores de los parámetros en la sentencia. Los valores se asignan a los marcadores de posición en el orden en el que aparecen en la consulta SQL. Por ejemplo:

```
sentencia.setString(1, "NuevoUsuario");

sentencia.setInt(2, 25);
```

3. Ejecutar la sentencia de inserción: Una vez que se han establecido los valores de los parámetros, puedes ejecutar la sentencia de inserción utilizando el método executeUpdate() de la sentencia. Este método devuelve el número de filas afectadas por la operación.

```
int filasAfectadas = sentencia.executeUpdate();
```

El valor de filasAfectadas te proporcionará información sobre si la inserción se realizó con éxito. Puedes verificar si es mayor que 0 para confirmar que se agregó al menos una fila a la tabla.

Actualizando filas

La actualización de filas en una base de datos es una operación común en aplicaciones JDBC. Permite modificar los datos existentes en una tabla. Aquí hay una descripción más detallada:

1. Preparar la sentencia de actualización: Para actualizar filas, se utiliza una PreparedStatement. Se crea una consulta SQL que especifica qué columnas se modificarán y se establece un criterio para identificar las filas que se actualizarán. Aquí hay un ejemplo:

```
String consultaSQL = "UPDATE usuarios SET edad = ? WHERE nombre = ?";

PreparedStatement sentencia = conexion.prepareStatement(consultaSQL);
```

2. Establecer valores de los parámetros: Luego, se deben establecer los valores de los parámetros en la sentencia. Generalmente, esto implica definir los nuevos valores para las columnas que se van a actualizar y proporcionar un criterio para identificar las filas que se modificarán. Por ejemplo:

```
sentencia.setInt(1, 30);  // Nuevo valor para la edad

sentencia.setString(2, "UsuarioAActualizar");  // Criterio para identificar la fila a actualizar
```

3. Ejecutar la sentencia de actualización:

Una vez que se han establecido los valores de los parámetros, se ejecuta la sentencia de actualización utilizando el método executeUpdate() de la sentencia. Este método devuelve el número de filas afectadas por la operación:

```
int filasAfectadas = sentencia.executeUpdate();
```

Eliminando filas:

La eliminación de filas en una base de datos es otra operación esencial. Permite eliminar registros de una tabla. Para eliminar filas en una base de datos, se siguen estos pasos:

1. Preparación de la sentencia de eliminación: Para eliminar filas, se utiliza una PreparedStatement. Se crea una consulta SQL que especifica la tabla y un criterio para identificar las filas que se eliminarán. Aquí hay un ejemplo:

```
String consultaSQL = "DELETE FROM usuarios WHERE nombre = ?";

PreparedStatement sentencia = conexion.prepareStatement(consultaSQL);
```

2. Establecimiento de valores de los parámetros: Luego, se establece el valor del parámetro en la sentencia, que se utiliza para identificar las filas que se eliminarán:

```
sentencia.setString(1, "UsuarioAEliminar");
```

3. Ejecución de la sentencia de eliminación: Una vez que se ha establecido el valor del parámetro, se puede ejecutar la sentencia de eliminación utilizando el método executeUpdate(). Esto proporcionará el número de filas afectadas:

```
int filasAfectadas = sentencia.executeUpdate();
```

Otras sentencias de modificación

1. Procedimientos almacenados:

JDBC permite ejecutar procedimientos almacenados en la base de datos. Para ello, se utiliza un objeto CallableStatement en lugar de una PreparedStatement. Se crea una consulta SQL que invoca el procedimiento almacenado y se establecen los valores de los parámetros (si el procedimiento los requiere). La ejecución del procedimiento se realiza con el método execute(), y se pueden recuperar resultados si el procedimiento los devuelve.

2. Transacciones:

JDBC facilita el control de transacciones en la base de datos. Puedes iniciar una transacción con el método setAutoCommit(false) en la conexión, ejecutar una serie

de operaciones y confirmar la transacción con commit() si todas las operaciones tienen éxito, o revertirla con rollback() en caso de errores. Esto garantiza la integridad de los datos y permite realizar cambios en la base de datos de manera atómica.

Result Set (Conjunto de Resultados):

Un "Result Set" es una estructura de datos proporcionada por JDBC (Java Database Connectivity) que representa los resultados de una consulta a una base de datos. En otras palabras, es el conjunto de filas que se obtiene como respuesta después de ejecutar una consulta SQL en una base de datos utilizando Java. El Result Set permite acceder y manipular los datos devueltos de manera programática.

A continuación, se presentan los aspectos clave del Result Set:

1. Creación del Result Set: Después de ejecutar una consulta SQL, se obtiene un Result Set a través de métodos como executeQuery() en un objeto Statement. Por ejemplo:

```
Statement statement = conexion.createStatement();

ResultSet resultSet = statement.executeQuery("SELECT * FROM usuarios");
```

2. Acceso a los datos: El Result Set se comporta como una tabla virtual que se puede recorrer fila por fila. Los métodos next() permiten avanzar a la siguiente fila, y luego se pueden obtener los valores de las columnas de esa fila mediante métodos como getString(), getInt(), etc.

```
while (resultSet.next()) {
    String nombre = resultSet.getString("nombre");
    int edad = resultSet.getInt("edad");
    // Acceder a otros valores de columnas
}
```

3. Manipulación de datos: Los Result Sets también permiten actualizar, eliminar y agregar filas a través de las operaciones de modificación adecuadas. Sin embargo, es importante tener en cuenta que no todos los Result Sets admiten estas operaciones, ya que depende del tipo de Result Set y la configuración de la base de datos.
4. Tipos de Result Set: JDBC proporciona varios tipos de Result Sets, como:

- Forward-Only Result Set: Permite moverse solo hacia adelante, desde la primera fila hasta la última.
- Scrollable Result Set: Permite moverse hacia adelante y hacia atrás a través de las filas.
- Updatable Result Set: Permite actualizar las filas del Result Set y reflejar los cambios en la base de datos.

5. Cierre del Result Set: Después de finalizar el uso del Result Set, es importante cerrarlo para liberar los recursos, lo que se hace llamando al método close().

```
resultSet.close();
```

El Result Set es una parte fundamental de JDBC que permite a las aplicaciones Java interactuar con bases de datos de manera eficiente. Proporciona una interfaz para acceder a los datos recuperados de consultas SQL y realizar diversas operaciones en ellos.

Mapeo entre SQL y Java Data Types

El mapeo entre los tipos de datos de SQL y los tipos de datos en Java es fundamental al interactuar con una base de datos utilizando JDBC (Java Database Connectivity). A continuación, se explica el mapeo común entre estos dos mundos:

1. VARCHAR / CHAR / TEXT en SQL a String en Java: Los tipos de datos de texto en SQL, como VARCHAR, CHAR y TEXT, generalmente se asignan al tipo de datos String en Java. Esto permite almacenar y recuperar cadenas de caracteres desde la base de datos.
2. INTEGER / SMALLINT / BIGINT en SQL a tipos numéricos en Java: Los tipos numéricos en SQL, como INTEGER, SMALLINT y BIGINT, se asignan a tipos numéricos en Java, como int, short o long, respectivamente.
3. DECIMAL / NUMERIC en SQL a BigDecimal en Java: Los tipos DECIMAL y NUMERIC en SQL, que se utilizan para valores numéricos de precisión fija, se asignan al tipo BigDecimal en Java debido a su capacidad de representar números con alta precisión.
4. DATE / TIME / TIMESTAMP en SQL a tipos de fecha y hora en Java: Los tipos de datos de fecha y hora en SQL, como DATE, TIME y TIMESTAMP, se asignan a tipos de fecha y hora en Java, como java.util.Date, java.sql.Time y java.sql.Timestamp.
5. BOOLEAN en SQL a boolean en Java: El tipo de datos BOOLEAN en SQL se asigna directamente al tipo de datos boolean en Java.

6. BLOB / CLOB en SQL a tipos de datos específicos en Java: Los tipos de datos binarios grandes (BLOB) y los tipos de datos de caracteres grandes (CLOB) en SQL se asignan a tipos de datos específicos en Java, como byte[] para BLOB y java.sql.Clob para CLOB.
7. ENUM en SQL a tipos enumerados en Java: Algunas bases de datos, como PostgreSQL, admiten tipos ENUM. En Java, se pueden mapear a enumerados personalizados que representan los valores permitidos del ENUM.
8. Otros tipos de datos especializados: Algunas bases de datos pueden admitir tipos de datos especializados, como geoespaciales o tipos personalizados. Estos tipos de datos pueden requerir mapeos personalizados a clases Java.

El mapeo entre SQL y Java data types es crucial para garantizar que los datos se almacenen y recuperen de manera adecuada y consistente al interactuar con una base de datos a través de JDBC. Es importante tener en cuenta que el mapeo puede variar ligeramente según la base de datos y el controlador JDBC utilizado, por lo que es esencial verificar la documentación correspondiente.

1	Text	`CHAR`
		`VARCHAR`
		`TEXT`
			`String`

2	Integer	`INTEGER`
		`NUMERIC`
		`BIT`[3]
			`int`

3	Decimal	`FLOAT`
		`DOUBLE`
		`REAL`
		`DECIMAL`
			`double`

4	Binary	`BLOB` `BIT`[4]	`byte[]`
5	Text Set	`SET`	`String[]`
6	Datetime	`DATETIME` `TIMESTAMP`	`String`
7	Date	`DATE`	`String`
8	Time	`TIME`	`String`
9	Interval	`INTERVAL`	`int`
10	Primary Key	`INTEGER`[5]	*reference*

Prepared Statement:

Un Prepared Statement es una representación precompilada de una consulta SQL que se envía a la base de datos con espacios reservados para parámetros. Los valores de los parámetros se establecen posteriormente, lo que permite una reutilización eficiente de la consulta con diferentes valores sin tener que volver a compilarla cada vez.

A continuación, se describen los aspectos clave de los Prepared Statements:

1. Creación de un Prepared Statement:

Para crear un Prepared Statement, se utiliza una instancia de la clase PreparedStatement que se obtiene a partir de un objeto Connection. La consulta SQL se pasa como un argumento a prepareStatement() con los marcadores de posición de parámetros (generalmente representados por signos de interrogación) que serán reemplazados posteriormente.

```
String consultaSQL = "SELECT * FROM usuarios WHERE nombre = ? AND edad = ?";

PreparedStatement preparedStatement = conexion.prepareStatement(consultaSQL);
```

2. Establecimiento de valores de parámetros:

Después de crear el Prepared Statement, se pueden establecer los valores de los parámetros utilizando los métodos setX() correspondientes, donde 'X' representa el tipo de dato del parámetro. Esto ayuda a evitar la inyección SQL y garantiza que los datos ingresados sean seguros.

```
preparedStatement.setString(1, "UsuarioEjemplo");

preparedStatement.setInt(2, 30);
```

3. Ejecución del Prepared Statement:

Una vez que se han establecido todos los valores de los parámetros, el Prepared Statement se ejecuta con los métodos executeQuery() o executeUpdate(), dependiendo de si la consulta es de lectura o de modificación, respectivamente.

```
ResultSet resultado = preparedStatement.executeQuery();

// O

int filasAfectadas = preparedStatement.executeUpdate();
```

4. Reutilización eficiente:

Una ventaja clave de los Prepared Statements es que pueden reutilizarse con diferentes valores de parámetros sin tener que recompilar la consulta cada vez. Esto mejora significativamente el rendimiento, ya que la base de datos solo debe compilar la consulta una vez.

5. Cierre del Prepared Statement:

Al igual que con otros objetos JDBC, es importante cerrar el Prepared Statement para liberar recursos una vez que ya no se necesite.

```
preparedStatement.close();
```

Angel Yeremi Ledesma Ayala

Los Prepared Statements son una herramienta fundamental en JDBC para ejecutar consultas SQL de manera segura y eficiente, y se utilizan comúnmente para interactuar con bases de datos en aplicaciones Java. Al evitar la inyección SQL y mejorar el rendimiento, proporcionan una forma segura y eficiente de acceder y modificar datos en una base de datos.

SQLException:

Un SQLException (Excepción SQL) es una excepción en Java que se utiliza para manejar errores relacionados con operaciones de bases de datos y consultas SQL. Esta excepción se genera cuando ocurre un problema durante la comunicación con una base de datos o cuando se ejecuta una consulta SQL. En tercera persona, se puede describir de la siguiente manera:

La SQLException es una clase de excepción en Java que se utiliza para gestionar situaciones excepcionales que surgen al interactuar con bases de datos a través de JDBC (Java Database Connectivity).

Los escenarios comunes en los que se genera una SQLException incluyen:

1. Errores de sintaxis SQL: Cuando una consulta SQL contiene errores de sintaxis, como un nombre de tabla incorrecto o una cláusula WHERE mal formada, se generará una SQLException.
2. Problemas de conexión: Si no se puede establecer una conexión con la base de datos o la conexión se interrumpe de manera inesperada, se generará una SQLException.
3. Errores de restricción de integridad: Cuando se intenta realizar una operación que viola restricciones de integridad de datos, como una clave primaria o una clave externa, se generará una SQLException.
4. Excepciones relacionadas con Prepared Statements: Al utilizar Prepared Statements, es posible que ocurran errores durante la configuración de los parámetros o la ejecución de la consulta, lo que dará lugar a una SQLException.

La clase SQLException proporciona información detallada sobre el error, incluyendo un código de error, una descripción del error y la ubicación en el código donde se generó la excepción. Esto permite a los programadores identificar y solucionar problemas de manera efectiva.